ScaleDux

TASK 2: Startup Pitch Text Evaluation with NLP

Mohan Raj Murugesan | rajm82776@gmail.com | LinkedIn: Mohan Raj

1. Problem Statement

Evaluating startup pitch decks is often subjective and inconsistent due to varied formats and content quality. This project aims to develop an AI-driven NLP pipeline to extract key investment signals and generate a standardized quality score for each deck.

2. Objective

- Extract structured information from real-world startup pitch decks (PDF format).
- Analyze core business dimensions such as Problem Clarity, Market Potential, Traction, Team Strength, and Business Model.
- Generate a composite quality score per deck to rank startups.

3. Dataset Description

The dataset consists of 5 real-world startup pitch decks in PDF format, shared by the company. These decks represent famous startups at early stages and cover a range of industries and business models.

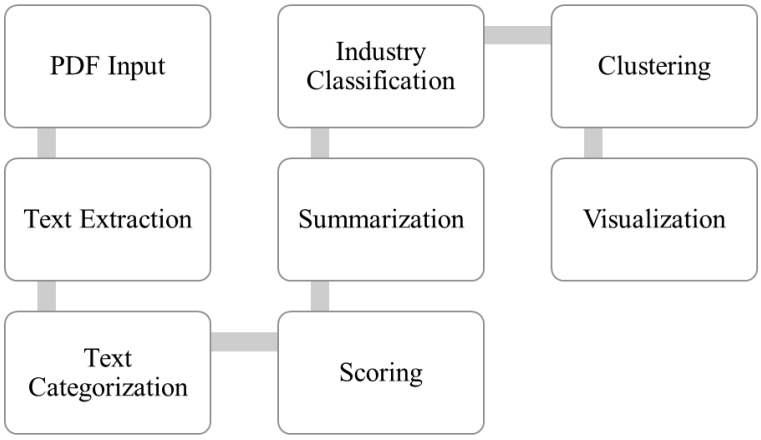
4. Software Dependencies

```
%pip install pdfplumber
%pip install vaderSentiment
%pip install dash
!pip install pdfplumber spacy vaderSentiment transformers pandas
plotly==5.15.0 kaleido==0.2.1 pytesseract
!apt-get install -y tesseract-ocr libtesseract-dev
!python -m spacy download en_core_web_sm
```

5. Approach

5.1 Pipeline Architecture

The system follows a sequential processing pipeline:



5.2 Methodological Framework

The approach combines multiple NLP techniques:

1. Hybrid Text Extraction: Combines text-based and OCR methods for comprehensive content capture
2. Rule-based Categorization: Uses keyword matching and Named Entity Recognition (NER)
3. Multi-modal Scoring: Integrates sentiment analysis and zero-shot classification
4. Transformer-based Analysis: Leverages pre-trained models for summarization and classification
5. Unsupervised Clustering: Groups similar pitch decks for comparative analysis

6. Implementation:

6.1 Text Extraction and Cleaning

A Text Extraction:

```
def extract_text_hybrid_easyocr(pdf_file):
    """Extract text from PDF using hybrid method (pdfplumber +
EasyOCR)."""
    text = []
    try:
```

```
        with pdfplumber.open(pdf_file) as pdf:
            for page in pdf.pages:
                page_text = page.extract_text()
                if page_text and len(page_text.strip()) > 10:
                    text.append(page_text.strip())
                else:
                    # Fallback to OCR (placeholder for EasyOCR)
                    try:
                        img = page.to_image(resolution=300).original
                        img = img.convert("L")  # Grayscale
                        page_text = pytesseract.image_to_string(img,
config='--psm 6 --oem 3')
                        text.append(page_text.strip())
                    except Exception as e:
                        logging.warning(f"OCR failed for {pdf_file} on
page {page.page_number}: {e}")
                        text.append("")
        return text if text else ["No text extracted"]
    except Exception as e:
        logging.error(f"Error processing {pdf_file}: {e}")
        return ["No text extracted"]
```

Extracts text from PDF pitch decks using a hybrid approach: pdfplumber for text-based PDFs and OCR (via pytesseract) as a fallback for image-based or text-poor pages.

Logic Breakdown

1. **Input**: Takes a pdf_file (path to a PDF) as input.
2. **Text Extraction with pdfplumber**:
    - Opens the PDF using pdfplumber.open in a context manager to ensure proper file handling.
    - Iterates through each page and attempts to extract text using page.extract_text().
    - Checks if the extracted text is valid (page_text exists and has >10 characters after stripping whitespace) to ensure meaningful content.
    - If valid, appends the stripped text to the text list.
3. **OCR Fallback**:
    - If pdfplumber fails (no text or <10 characters), converts the page to an image (page.to_image(resolution=300)).
    - Converts the image to grayscale (img.convert("L")) to improve OCR accuracy by reducing noise.
    - Uses pytesseract.image_to_string with configuration --psm 6 --oem 3:

        - Assumes a single uniform block of text, suitable for pitch deck slides with varied layouts.
        - Uses Tesseract's default OCR engine mode for flexibility.
    - Appends the OCR-extracted text (stripped) to the text list.
    - If OCR fails, logs a warning and appends an empty string.
4. **Error Handling**:
    - Wraps the entire process in a try-except block to catch PDF processing errors (e.g., corrupted files).
    - Logs errors with logging.error and returns ["No text extracted"] as a fallback.
5. **Output**: Returns a list of page texts or ["No text extracted"] if no text is retrieved.

B Cleaning:

```
def clean_text(text):
    """Clean text for summarization."""
    text = re.sub(r"\n\s*\n", "\n", text)
    text = re.sub(r"\d+/\d+", "", text)
    text = re.sub(r"(\bpage\b|\b\d+\b)", "", text, flags=re.IGNORECASE)
    return text.strip()
```

Cleans raw text extracted from PDFs to prepare it for downstream NLP tasks, such as summarization or scoring, by removing noise and irrelevant content.

Logic Breakdown

1. **Input**: Takes a string (text) extracted from a PDF page or deck.
2. **Cleaning Steps**:
    - **Remove Multiple Newlines**: re.sub(r"\n\s*\n", "\n", text) replaces multiple consecutive newlines (with optional whitespace) with a single newline to reduce formatting artifacts.
    - **Remove Fractions/Dates**: re.sub(r"\d+/\d+", "", text) removes patterns like "1/2" or "2023/24", which could represent fractions, dates, or slide numbers, as these are irrelevant for summarization.
    - **Remove Page Numbers and "page"**: re.sub(r"(\bpage\b|\b\d+\b)", "", text, flags=re.IGNORECASE) removes the word "page" and standalone numbers (e.g., page numbers), ignoring case sensitivity.
3. **Output**: Returns the cleaned text, stripped of leading/trailing whitespace

6.2 Truncate

```
def truncate_to_max_tokens(text, max_tokens=512):
    """Truncate text to fit within max_tokens for the model."""
    tokens = tokenizer(text, truncation=True, max_length=max_tokens,
return_tensors="pt")
```

```
    truncated_text = tokenizer.decode(tokens["input_ids"][0],
skip_special_tokens=True)
    return truncated_text
```

The truncate_to_max_tokens function ensures that input text does not exceed the maximum token limit of the NLP model (e.g., BART or T5) used in the pipeline, preventing errors during processing and optimizing performance.

Logic Breakdown

1. **Input**:
    - text: A string (e.g., extracted from a pitch deck via extract_text_hybrid_easyocr).
    - max_tokens: The maximum number of tokens allowed (default is 512, a common limit for transformer models like BART and T5).
2. **Tokenization**:
    - Uses the tokenizer (previously initialized as AutoTokenizer.from_pretrained("facebook/bart-large-mnli")) to tokenize the input text.
    - Parameters:
        - truncation=True: Automatically truncates the text to fit within max_tokens.
        - max_length=max_tokens: Sets the maximum token length (e.g., 512).
        - return_tensors="pt": Returns the output as PyTorch tensors, compatible with transformer models.
    - The tokenizer splits the text into tokens (subword units) based on the model's vocabulary and handles special tokens (e.g., <s>, </s>).
3. **Decoding**:
    - Converts the tokenized input IDs (tokens["input_ids"][0]) back to a string using tokenizer.decode.
    - skip_special_tokens=True: Excludes special tokens (e.g., <s>, </s>) from the output, ensuring the returned text is human-readable.
4. **Output**:
    - Returns the truncated_text, a string containing the first max_tokens tokens' worth of content, decoded back to natural language.

6.3 Text Categorization

```
def categorize_text(pages):

    """Categorize text into sections using keywords and spaCy entities."""

    sections = {

        "Problem": ["problem", "challenge", "pain point", "issue", "need",
"obstacle", "difficulty", "barrier"],
```

```
        "Solution": ["solution", "product", "offering", "solve",
"platform", "service", "technology", "app"],

        "Market": ["market", "TAM", "SAM", "SOM", "opportunity",
"industry", "segment", "potential", "demand"],

        "Traction": ["traction", "growth", "users", "revenue", "metrics",
"customers", "sales", "adoption", "engagement"],

        "Team": ["team", "founder", "experience", "background", "advisor",
"staff", "leadership", "executive", "member"],

        "Business Model": ["business model", "revenue", "monetization",
"pricing", "subscription", "income", "profit", "model"],

        "Vision/Moat": ["vision", "moat", "advantage", "IP", "patent",
"defensibility", "strategy", "differentiation", "unique"]

    }

    categorized = {key: [] for key in sections}

    for page in pages:

        if page == "No text extracted":

            continue

        doc = nlp(page)

        current_section = None

        for line in page.split("\n"):

            line_lower = line.lower()

            for section, keywords in sections.items():

                if any(keyword in line_lower for keyword in keywords):

                    current_section = section

                    break

            if current_section:
```

```
                categorized[current_section].append(line)

        for ent in doc.ents:

            if ent.label_ in ["PERSON", "ORG"] and any(keyword in
page.lower() for keyword in sections["Team"]):

                categorized["Team"].append(ent.text)

            elif ent.label_ in ["MONEY", "CARDINAL"] and any(keyword in
page.lower() for keyword in sections["Market"]):

                categorized["Market"].append(ent.text)

    return {key: " ".join(set(val)) for key, val in categorized.items() if
val}
```

The categorize_text function processes extracted text from pitch deck pages (e.g., from extract_text_hybrid_easyocr) and organizes it into predefined sections (Problem, Solution, Market, etc.) using a combination of keyword matching and spaCy's named entity recognition (NER). This structured output facilitates downstream tasks like scoring (BART) and summarization (T5).

Logic Breakdown

1. **Input**:
    - pages: A list of strings, each representing the text extracted from a PDF page (e.g., output from extract_text_hybrid_easyocr).
2. **Section Definitions**:
    - Defines a dictionary sections mapping section names (e.g., "Problem", "Solution") to lists of relevant keywords (e.g., ["problem", "challenge", "pain point"] for "Problem").
    - These keywords are tailored to common pitch deck sections, reflecting investor-relevant topics.
3. **Initialization**:
    - Creates a dictionary categorized with empty lists for each section to store matching text.
4. **Keyword-Based Categorization**:
    - Iterates through each page in pages.
    - Skips pages with "No text extracted" (from failed extraction).
    - Processes the page with spaCy (nlp(page)) to create a doc object for entity recognition.
    - Splits the page into lines (page.split("\n")) for granular analysis.

- For each line:
  - Converts the line to lowercase (line_lower) for case-insensitive keyword matching.
  - Checks if any keyword from a section's keyword list appears in the line.
  - If a match is found, assigns the line to the corresponding section (current_section) and appends it to categorized[current_section].
  - Breaks after the first match to avoid assigning a line to multiple sections.

5. **Entity-Based Categorization**:
   - Uses spaCy's NER to identify entities in the page (doc.ents).
   - Assigns entities to sections based on their type and context:
     - If the entity is a "PERSON" or "ORG" and the page contains "Team" keywords, appends the entity text to the "Team" section.
     - If the entity is "MONEY" or "CARDINAL" and the page contains "Market" keywords, appends the entity text to the "Market" section.
6. **Output**:
   - Converts each section's list of lines/entities to a single string using " ".join(set(val)), removing duplicates (set(val)) to avoid redundant text.
   - Returns a dictionary of non-empty sections, with keys as section names and values as concatenated text.

6.4 Scoring Each Deck

6.4.1 score_dimension Function

```
def score_dimension(text, dimension, criteria):
    """Score a dimension using NLP."""
    if not text or text == "No text extracted":
        return 0
    sentiment = vader.polarity_scores(text[:500])["compound"]
    try:
        scores = zero_shot(text[:500], candidate_labels=criteria,
multi_label=False)
        quality_score = scores["scores"][0] * 10
    except Exception as e:
        print(f"Zero-shot classification failed for {dimension}: {e}")
        quality_score = 0
    return round(0.7 * quality_score + 0.3 * (sentiment + 1) * 5, 1)
```

Scores a single pitch deck section (e.g., Problem, Market) based on its text content, combining a quality score (from BART zero-shot classification) and a sentiment score (from VADER).

Logic Breakdown

1. **Input**:
    - text: The text for a specific section (e.g., from categorize_text).
    - dimension: The section name (e.g., "Problem").
    - criteria: A list of labels for zero-shot classification (e.g., ["clear and specific", "vague", "generic"] for Problem).
2. **Error Handling**:
    - If text is empty or equals "No text extracted" (from extract_text_hybrid_easyocr), returns a score of 0 to handle missing or invalid input.
3. **Sentiment Analysis (VADER)**:
    - Uses VADER's polarity_scores to compute a sentiment score for the first 500 characters of the text (text[:500]).
    - Extracts the compound score, which ranges from -1 (negative) to +1 (positive).
    - **Purpose**: Measures the tone (e.g., confident, optimistic) of the section, as a positive tone is appealing to investors.
    - **Why 500 characters?**: Limits input to avoid performance issues with VADER and focuses on the most relevant text (pitch deck sections are typically concise).
4. **Zero-Shot Classification (BART)**:
    - Uses the BART model (facebook/bart-large-mnli, initialized as zero_shot) to perform zero-shot classification on the first 500 characters of the text.
    - candidate_labels=criteria: Evaluates how well the text matches the provided labels (e.g., "clear and specific" vs. "vague").
    - multi_label=False: Selects the most likely label, returning a probability score for each label in criteria.
    - Takes the score for the first label (scores["scores"][0]), assumed to be the positive criterion (e.g., "clear and specific"), and scales it to a 0–10 range: quality_score=scores["scores"][0]×10
    - If classification fails (e.g., due to model errors), logs the error and sets quality_score = 0.
5. **Combined Score Calculation**:
    - Combines the quality_score (0–10) and sentiment (-1 to +1) into a single score: dimension_score=0.7×quality_score+0.3×(sentiment+1)×5
    - **Sentiment Normalization**:
        - sentiment + 1: Shifts the sentiment range from [-1, +1] to [0, 2].
        - (sentiment + 1) * 5: Scales it to [0, 10] for consistency with quality_score.
    - **Weighting**:
        - 70% weight on quality_score (0.7): Emphasizes content quality (e.g., clarity, specificity).
        - 30% weight on sentiment (0.3): Reflects the importance of tone but prioritizes content.
    - **Rounding**: Rounds the result to one decimal place for readability.

6. **Output**:
    - Returns a single score (0–10) for the dimension, reflecting both content quality and sentiment.

6.4.2 score_deck Function

```
def score_deck(sections):
    """Score a deck across all dimensions."""
    criteria = {
        "Problem": ["clear and specific", "vague", "generic"],
        "Market": ["large and quantified", "small", "unfocused"],
        "Traction": ["strong metrics", "weak metrics", "no data"],
        "Team": ["experienced and relevant", "inexperienced", "generic"],
        "Business Model": ["clear monetization", "unclear",
"unsustainable"],
        "Vision/Moat": ["defensible and scalable", "generic", "weak"]
    }
    scores = {}
    for dim in criteria:
        scores[dim] = score_dimension(sections.get(dim, ""), dim,
criteria[dim])
    overall_text = " ".join(sections.values())
    scores["Confidence"] =
round(vader.polarity_scores(overall_text)["compound"] * 5 + 5, 1) if
overall_text else 0
    total = sum(scores.values())
    normalized = round((total / 70) * 100, 1)
    return scores, normalized
```

Scores an entire pitch deck across multiple dimensions (Problem, Market, etc.) and computes a normalized final score, including a Confidence score for the deck's overall tone.

Logic Breakdown

1. **Input**:
    - sections: A dictionary of section texts (e.g., from categorize_text), with keys like "Problem", "Market", and values as concatenated text.
2. **Criteria Definitions**:
    - Defines a dictionary criteria mapping each dimension to a list of zero-shot labels (e.g., "Problem": ["clear and specific", "vague", "generic"]).
    - The first label is positive (e.g., "clear and specific"), followed by negative labels (e.g., "vague", "generic").
3. **Dimension Scoring**:

- Iterates through each dimension in criteria.
- Calls score_dimension with the section's text (sections.get(dim, "")), dimension name, and corresponding criteria.
- Stores each dimension's score (0–10) in the scores dictionary.
- Uses sections.get(dim, "") to handle missing sections, defaulting to an empty string (which returns a score of 0).

4. **Confidence Score**:
   - Concatenates all section texts into overall_text (" ".join(sections.values())).
   - Computes a Confidence score using VADER's compound score for the entire deck:

     $$Confidence = vader.polarity\_scores(overall\_text)["compound"] \times 5 + 5$$

   - **Normalization**:
     - compound ranges from -1 to +1.
     - Multiplying by 5 scales it to [-5, +5].
     - Adding 5 shifts it to [0, 10], aligning with other dimension scores.
   - If overall_text is empty, sets Confidence to 0.
   - Rounds to one decimal place.
5. **Total and Normalized Score**:
   - Sums all scores (six dimensions + Confidence) to get total:

     $$total = \sum dimscores[dim] + scores["Confidence"]$$

   - Normalizes the total to a 0–100 scale, assuming a maximum of 10 per dimension for seven dimensions (6 + Confidence):

     $$normalized = (total 70) \times 100$$

   - Rounds to one decimal place.
6. **Output**:
   - Returns a tuple: scores (dictionary of dimension scores, including Confidence) and normalized (final score on a 0–100 scale).

6.4.3 How Scoring Is Done

1. **Per-Dimension Scoring (score_dimension)**:
   - **Input**: Text for a section (e.g., Problem text from categorize_text).
   - **Process**:
     - **Sentiment**: VADER computes a compound score (-1 to +1) for the first 500 characters, reflecting tone.
     - **Quality**: BART evaluates the first 500 characters against criteria (e.g., "clear and specific"), producing a probability (0–1) scaled to 0–10.
     - **Combined Score**:

       $$dimension\_score = 0.7 \times (BART\ score \times 10) + 0.3 \times (VADER\ compound + 1) \times 5$$

        - Weights prioritize quality (70%) over sentiment (30%).
    - **Output**: A score (0–10) for the dimension.
    - **Example**:
        - Text: "Our platform solves inefficiencies in healthcare delivery."
        - VADER: compound = 0.4 → (0.4 + 1) * 5 = 7.0
        - BART: scores["clear and specific"] = 0.8 → 0.8 * 10 = 8.0
        - Score: 0.7×8.0+0.3×7.0=5.6+2.1=7.7
2. **Deck Scoring (score_deck)**:
    - **Input**: Dictionary of section texts from categorize_text.
    - **Process**:
        - Scores six dimensions (Problem, Market, Traction, Team, Business Model, Vision/Moat) using score_dimension.
        - Computes a Confidence score for the entire deck's text: *Confidence=(VADER compound×5)+5*
        - Sums all seven scores (6 dimensions + Confidence) to get total (0–70).
        - Normalizes to 0–100: *normalized=(total/70)×100*
    - **Output**: A dictionary of scores and a normalized final score.
    - **Example**:
        - Scores: Problem = 7.7, Market = 8.0, Traction = 6.5, Team = 7.0, Business Model = 6.8, Vision/Moat = 7.2, Confidence = 6.0.
        - Total: 7.7+8.0+6.5+7.0+6.8+7.2+6.0=49.2
        - Normalized: (49.2/70)×100=70.3

6.4.4 How the Final Score Is Calculated

The final score (normalized) is a normalized percentage (0–100) based on the sum of seven scores (six dimensions + Confidence), each ranging from 0–10.

1. **Dimension Scores**:
    - Each of the six dimensions (Problem, Market, Traction, Team, Business Model, Vision/Moat) is scored using score_dimension:

      dimension_score=0.7×(BART score×10)+0.3×(VADER compound+1)×5

    - Range: 0–10 per dimension.
    - Total for six dimensions: 0–60.
2. **Confidence Score**:
    - Computed for the entire deck's text: *Confidence=(VADER compound×5)+5*
    - Range: 0–10 (since compound is -1 to +1, scaled to 0–10).
3. **Total Score**:
    - Sum of all seven scores: total=Problem+Market+Traction+Team+Business Model+Vision/Moat+Confidence
    - Maximum: 10×7=70

4. **Normalized Final Score**:
    - Scaled to 0–100: *normalized=(total/70)×100*
    - Rounded to one decimal place for readability.
5. **Example Calculation**:
    - Assume scores: Problem = 7.7, Market = 8.0, Traction = 6.5, Team = 7.0, Business Model = 6.8, Vision/Moat = 7.2, Confidence = 6.0.
    - Total: 7.7+8.0+6.5+7.0+6.8+7.2+6.0=49.2
    - Normalized: (49.2/70)×100=70.3Output: scores = {"Problem": 7.7, "Market": 8.0, ..., "Confidence": 6.0}, normalized = 70.3.

Final Score Calculation:

$$total=\sum_{6dimensions}(0.7\times BART\times 10+0.3\times(VADER+1)\times 5)+(VADERdeck\times 5+5)$$

$$normalized=(total/70)\times 100$$

6.5 Processing of all Decks

```
def process_decks(folder_path, output_folder):
    """Process all PDFs, extract text, and analyze."""
    os.makedirs(output_folder, exist_ok=True)
    deck_names = [
        "Pitch-Example-Air-BnB-PDF",
        "uber-pitch-deck",

"6737d05825e11f73f6d5a289_Ndc8GMUtaMNHOXDfqftyW1Jb7b5h2JE_ThY_Joc5Cf8",
        "FACEBOOK",
        "doordash-pitch-deck"
    ]
    results = []

    for deck_name in deck_names:
        pdf_file = Path(folder_path) / f"{deck_name}.pdf"
        txt_out_path = Path(output_folder) / f"{deck_name}.txt"

        if not pdf_file.exists():
            logging.warning(f"File {pdf_file} not found")
            text = ["No text extracted"]
        else:
            text = extract_text_hybrid_easyocr(pdf_file)
            with open(txt_out_path, "w", encoding="utf-8") as f:
                for i, slide in enumerate(text):
```

```
                    f.write(f"---SLIDE {i+1}---\n{slide}\n\n")

        sections = categorize_text(text)
        scores, final_score = score_deck(sections)

        try:
            clean_summary_text = clean_text(" ".join(text)[:1000])
            summary = summarizer(clean_summary_text, max_length=50,
min_length=20, do_sample=False, max_new_tokens=None)[0]["summary_text"]
            insight = f"Insight: {summarizer(clean_summary_text[:200],
max_length=10, min_length=5, do_sample=False,
max_new_tokens=None)[0]['summary_text']}"
        except Exception as e:
            logging.error(f"Error summarizing {deck_name}: {e}")
            summary = f"Summary not available for {deck_name}"
            insight = f"Insight not available for {deck_name}"

        try:
            full_text = " ".join(text)
            classification_input = truncate_to_max_tokens(full_text,
max_tokens=512)
            candidate_labels = ["Fintech", "HealthTech", "SaaS", "B2C",
"Social Media", "Food Delivery", "Ride-Sharing"]
            industry = zero_shot(classification_input,
candidate_labels=candidate_labels, multi_label=False)
            industry_label =
industry["labels"][industry["scores"].index(max(industry["scores"]))]
            # Fallback: Keyword-based classification
            if industry_label not in ["Fintech", "HealthTech", "SaaS",
"B2C", "Social Media", "Food Delivery", "Ride-Sharing"]:
                doc = nlp(full_text)
                keywords = {
                    "Fintech": [("payment", 2), ("finance", 1.5),
("banking", 1.5), ("transaction", 1)],
                    "HealthTech": [("health", 2), ("medical", 1.5),
("patient", 1), ("care", 1)],
                    "SaaS": [("software", 2), ("subscription", 1.5),
("cloud", 1), ("tool", 1)],
                    "B2C": [("consumer", 2), ("marketplace", 1.5),
("booking", 1), ("sharing", 1)],
```

```
                    "Social Media": [("social", 2), ("network", 1.5),
("connect", 1), ("community", 1)],
                    "Food Delivery": [("delivery", 2), ("restaurant",
1.5), ("food", 1.5), ("courier", 1)],
                    "Ride-Sharing": [("ride", 2), ("transport", 1.5),
("driver", 1), ("car", 1)]
                }
                keyword_scores = {label: 0 for label in candidate_labels}
                for label, kws in keywords.items():
                    for kw, weight in kws:
                        keyword_scores[label] += weight *
full_text.lower().count(kw)
                for ent in doc.ents:
                    if ent.label_ in ["ORG", "PRODUCT"]:
                        for label, kws in keywords.items():
                            if any(kw[0] in ent.text.lower() for kw in
kws):
                                keyword_scores[label] += 2
                max_score = max(keyword_scores.values(), default=0)
                if max_score > 0:
                    industry_label = max(keyword_scores,
key=keyword_scores.get)
        except Exception as e:
            logging.error(f"Industry classification failed for
{deck_name}: {e}")
            industry_label = "Unknown"

        # Improvement suggestions
        suggestions = []
        for dim, score in scores.items():
            if score < 5:
                suggestions.append(f"Improve {dim}: Ensure clear, specific
details and strong metrics.")

        results.append({
            "Deck": deck_name,
            **scores,
            "Final Score": final_score,
            "Summary": summary,
            "Insight": insight,
```

```
            "Industry": industry_label,
            "Suggestions": "; ".join(suggestions) if suggestions else "No
major improvements needed"
        })

    df = pd.DataFrame(results)
    # Clustering decks
    if len(df) > 1:
        dimensions = ["Problem", "Market", "Traction", "Team", "Business
Model", "Vision/Moat", "Confidence"]
        X = df[dimensions].fillna(0)
        kmeans = KMeans(n_clusters=min(3, len(df)), random_state=42)
        df["Cluster"] = kmeans.fit_predict(X)
    return df
```

Processes a predefined list of pitch deck PDFs, extracting text, categorizing it, scoring it, summarizing it, classifying the industry, generating suggestions, and clustering decks. Returns a DataFrame with comprehensive results.

Logic Breakdown

1. **Setup**:
    - Creates the output_folder if it doesn't exist (os.makedirs(output_folder, exist_ok=True)).
    - Defines a hardcoded list of deck_names (e.g., "Pitch-Example-Air-BnB-PDF", "uber-pitch-deck").
    - Initializes an empty results list to store analysis for each deck.
2. **Text Extraction**:
    - Constructs the PDF file path (pdf_file) for each deck using Path.
    - Checks if the PDF exists; if not, logs a warning and sets text = ["No text extracted"].
    - Otherwise, calls extract_text_hybrid_easyocr to extract text per page.
    - Saves extracted text to a .txt file in output_folder, with each slide labeled (e.g., ---SLIDE 1---).
3. **Text Categorization**:
    - Calls categorize_text to organize extracted text into sections (e.g., Problem, Market).

4. **Scoring**:
    - Calls score_deck to compute dimension scores (Problem, Market, Traction, Team, Business Model, Vision/Moat, Confidence) and a normalized final score (0–100).
    - **Final Score Calculation** (from score_deck):
        - Each dimension score (0–10) combines BART zero-shot quality (70%) and VADER sentiment (30%):

          $dimension\_score = 0.7 \times (BART\ score \times 10) + 0.3 \times (VADER\ compound + 1) \times 5$

        - Confidence score (0–10): $Confidence = (VADER\ compound \times 5) + 5$
        - Total score (0–70): $total = \sum_{6 dimensions} dimension\_score + Confidence$
        - Normalized final score (0–100): $final\_score = (total/70) \times 100$

      Example:

        - Scores: Problem = 7.7, Market = 8.0, Traction = 6.5, Team = 7.0, Business Model = 6.8, Vision/Moat = 7.2, Confidence = 6.0.
        - Total: $7.7+8.0+6.5+7.0+6.8+7.2+6.0=49.2$
        - Final Score:$(49.2/70) \times 100 = 70.3$
5. **Summarization**:
    - Concatenates all page texts, cleans the first 1000 characters (clean_text), and uses T5 (summarizer) to generate:
        - A summary (20–50 tokens).
        - An insight (5–10 tokens, based on the first 200 characters).
    - Handles errors by logging and setting fallback messages.
6. **Industry Classification**:
    - Concatenates all text, truncates to 512 tokens (truncate_to_max_tokens), and uses BART (zero_shot) to classify the deck's industry (e.g., Fintech, Social Media).
    - **Fallback**: If the predicted label isn't in the candidate list, uses a keyword-based approach:
        - Counts weighted keywords (e.g., "payment" for Fintech) in the text.
        - Boosts scores with spaCy NER (e.g., ORG, PRODUCT entities).
        - Selects the industry with the highest score.
    - Sets "Unknown" if classification fails.
7. **Improvement Suggestions**:
    - For each dimension with a score < 5, adds a suggestion to improve clarity or metrics.

- Joins suggestions with semicolons or sets a default message if none apply.

8. **Results Storage**:
   - Appends a dictionary to results with:
     - Deck name, dimension scores, final score, summary, insight, industry, suggestions.
9. **Clustering**:
   - Converts results to a DataFrame (df).
   - If multiple decks exist, applies KMeans clustering (up to 3 clusters) on dimension scores, filling missing values with 0.
   - Adds a "Cluster" column to group similar decks.
10. **Output**:
    - Returns the DataFrame with all results.

6.6 Visualization

6.6.1 Charts

```
def create_radar_chart(df, output_folder):
    """Create a radar chart comparing decks and save to file."""
    dimensions = ["Problem", "Market", "Traction", "Team", "Business
Model", "Vision/Moat", "Confidence"]
    fig = go.Figure()
    for _, row in df.iterrows():
        fig.add_trace(go.Scatterpolar(
            r=[row[dim] for dim in dimensions],
            theta=dimensions,
            fill="toself",
            name=row["Deck"]
        ))
    fig.update_layout(
        polar=dict(radialaxis=dict(visible=True, range=[0, 10])),
        showlegend=True,
        title="Pitch Deck Comparison (Radar Chart)"
    )
    fig.show()
    # Save to file
    output_path = os.path.join(output_folder, "radar_chart.png")
    try:
        fig.write_image(output_path)
        logging.info(f"Radar chart saved to {output_path}")
```

```
    except Exception as e:
        logging.error(f"Failed to save radar chart: {e}")
    return fig

def create_bar_chart(df, output_folder):
    """Create a bar chart of final scores and save to file."""
    fig = px.bar(df, x="Deck", y="Final Score", title="Final Scores by
Deck", color="Deck")
    fig.show()
    # Save to file
    output_path = os.path.join(output_folder, "bar_chart.png")
    try:
        fig.write_image(output_path)
        logging.info(f"Bar chart saved to {output_path}")
    except Exception as e:
        logging.error(f"Failed to save bar chart: {e}")
    return fig

def create_heatmap(df, output_folder):
    """Create a correlation heatmap of dimensions and save to file."""
    dimensions = ["Problem", "Market", "Traction", "Team", "Business
Model", "Vision/Moat", "Confidence"]
    corr = df[dimensions].corr()
    fig = px.imshow(corr, text_auto=True, title="Dimension Correlation
Heatmap")
    fig.show()
    # Save to file
    output_path = os.path.join(output_folder, "heatmap.png")
    try:
        fig.write_image(output_path)
        logging.info(f"Heatmap saved to {output_path}")
    except Exception as e:
        logging.error(f"Failed to save heatmap: {e}")
    return fig

def create_word_cloud(df, output_folder):
    """Create a word cloud from industry labels and save to file."""
    if "Industry" not in df.columns or df["Industry"].isna().all():
        logging.warning("No industry labels available for word cloud")
        return None
```

```
    # Create frequency dictionary of industry labels
    industry_counts = df["Industry"].value_counts().to_dict()
    # Generate word cloud with frequencies
    wordcloud = WordCloud(
        width=800,
        height=400,
        background_color="white",
        min_font_size=10,
        max_font_size=100
    ).generate_from_frequencies(industry_counts)
    # Convert to Plotly figure
    fig = go.Figure()
    plt.figure(figsize=(10, 5))
    plt.imshow(wordcloud, interpolation="bilinear")
    plt.axis("off")
    plt.title("Word Cloud of Industry Labels")
    buf = io.BytesIO()
    plt.savefig(buf, format="png")
    buf.seek(0)
    img_str = "" +
base64.b64encode(buf.read()).decode()
    # Save word cloud as image
    output_path = os.path.join(output_folder, "word_cloud.png")
    try:
        wordcloud.to_file(output_path)
        logging.info(f"Word cloud saved to {output_path}")
    except Exception as e:
        logging.error(f"Failed to save word cloud: {e}")
    buf.close()
    plt.close()
    fig.add_layout_image(
        dict(
            source=img_str,
            xref="paper", yref="paper",
            x=0, y=1,
            sizex=1, sizey=1,
            xanchor="left", yanchor="top",
            layer="below"
        )
    )
```

```
    fig.update_layout(
        title="Word Cloud of Industry Labels",
        xaxis=dict(visible=False),
        yaxis=dict(visible=False),
        width=800,
        height=400
    )
    fig.show()
    return fig
```

The visualization functions (create_radar_chart, create_bar_chart, create_heatmap, create_word_cloud) take the DataFrame output from process_decks and create:

1. **Radar Chart**: Compares dimension scores (Problem, Market, etc.) across decks.
2. **Bar Chart**: Displays final scores for each deck.
3. **Heatmap**: Shows correlations between dimension scores.
4. **Word Cloud**: Visualizes the frequency of industry labels.

These visualizations enhance interpretability of the pitch deck analysis, leveraging the final score (normalized 0–100 from score_deck) and dimension scores to provide insights for investors.

6.6.2 Dashboard

```
def create_dashboard(df, output_folder):
    """Create an interactive Dash dashboard."""
    app = dash.Dash(__name__)
    word_cloud_fig = create_word_cloud(df, output_folder)
    app.layout = html.Div([
        html.H1("Pitch Deck Evaluation Dashboard"),
        html.H2("Scores Table"),
        dcc.Graph(figure=create_radar_chart(df, output_folder)),
        dcc.Graph(figure=create_bar_chart(df, output_folder)),
        dcc.Graph(figure=create_heatmap(df, output_folder)),
        html.H2("Word Cloud of Industry Labels"),
        dcc.Graph(figure=word_cloud_fig) if word_cloud_fig else html.P("No
word cloud available"),
        html.H2("Detailed Results"),
        html.Table([
            html.Thead(html.Tr([html.Th(col) for col in df.columns])),
```

```
            html.Tbody([
                html.Tr([html.Td(df.iloc[i][col]) for col in df.columns])
                for i in range(len(df))
            ])
        ])
    ])
    return app
```

The create_dashboard function creates an interactive web-based dashboard using Dash, a Python framework for building analytical applications. The dashboard consolidates:

- **Visualizations**: Radar chart, bar chart, heatmap, and word cloud from the respective visualization functions.
- **Table**: A detailed table displaying all columns from the process_decks DataFrame (e.g., Deck, dimension scores, Final Score, Summary, Insight, Industry, Suggestions, Cluster).
- **Purpose**: Provides a user-friendly interface for investors to explore pitch deck analysis results, leveraging the final score (0–100) and dimension scores (0–10) computed by score_deck and processed by process_decks

6.7 Main Execution Block

```
if __name__ == "__main__":
    folder_path = "/content/pdf_decks"
    output_folder = "/content/output"
    df = process_decks(folder_path, output_folder)
    # Save DataFrame to CSV
    csv_path = os.path.join(output_folder, "results.csv")
    df.to_csv(csv_path, index=False)
    logging.info(f"DataFrame saved to {csv_path}")
    print("=== Pitch Deck Evaluation Dashboard ===")
    print(df)
    print("\nTop 3 Decks:")
    print(df.nlargest(3, "Final Score")[["Deck", "Final Score",
"Industry"]])
    print("\nBottom 3 Decks:")
    print(df.nsmallest(3, "Final Score")[["Deck", "Final Score",
"Industry"]])
    print("\n=== Summaries and Insights ===")
```

```
    for _, row in df.iterrows():
        print(f"Deck: {row['Deck']} ({row['Industry']})")
        print(f"Final Score: {row['Final Score']}")
        print(f"Insight: {row['Insight']}")
        summary_formatted = row["Summary"].replace(". ", ".\n- ")
        print(f"Summary:\n- {summary_formatted}")
        print()
    app = create_dashboard(df,output_folder)
    app.run(debug=True)
    if files is not None:  # Running in Colab
        files.download(csv_path)
        files.download(os.path.join(output_folder, "radar_chart.png"))
        files.download(os.path.join(output_folder, "bar_chart.png"))
        files.download(os.path.join(output_folder, "heatmap.png"))
        files.download(os.path.join(output_folder, "word_cloud.png"))
```

Executes the entire pitch deck analysis pipeline, saves results, prints key metrics, and launches an interactive dashboard, with optional file downloads for Colab.

7. Output Analysis

7.1 Results Summary

Based on the analysis of 5 pitch decks, the system generated the following results:

Deck	Problem	Market	Traction	Team	Business Model	Vision/Moat	Confidence	Final Score	Suggestions	Cluster
Pitch-Example-Air-BnB-PDF	6.4	9	7.7	0	5	7.4	9.4	64.1	Improve Team: Ensure clear, specific details and strong metrics.	2
uber-pitch-deck	6.2	6.8	8.4	9.5	7.5	5.5	10	77	No major improvements needed	0
6737d05825e11f73f6d5a289_Ndc8GMUtaMNHOXDfqftyW1Jb7b5h2JE_ThY_Joc5Cf8	6.4	7.8	0	0	0	0	1.6	22.6	Improve Traction: Ensure clear, specific details and strong metrics.; Improve Team: Ensure clear, specific details and strong metrics.; Improve Business Model: Ensure clear, specific details and strong metrics.; Improve Vision/Moat: Ensure clear, specific details and strong metrics.; Improve Confidence: Ensure clear, specific details and strong metrics.	2
FACEBOOK	0	6	8.6	7.4	6.2	4	10	60.3	Improve Problem: Ensure clear, specific details and strong metrics.; Improve Vision/Moat: Ensure clear, specific details and strong metrics.	0
doordash-pitch-deck	0	0	8.6	0	0	0	8.5	24.4	Improve Problem: Ensure clear, specific details and strong metrics.; Improve Market: Ensure clear, specific details and strong metrics.; Improve Team: Ensure clear, specific details and strong metrics.; Improve Business Model: Ensure clear, specific details and strong metrics.; Improve Vision/Moat: Ensure clear, specific details and strong metrics.	1

7.2 Visualization Insights

7.2.1 Radar Chart Analysis

The radar chart reveals distinct patterns:

- **Uber** shows the most balanced profile with consistently high scores

- **Airbnb** demonstrates strong market focus but lacks team information
- **Facebook** and others show significant gaps in key areas

Fig 7.1 Radar chart comparing decks

7.2.2 Correlation Heatmap Findings

The heatmap indicates:

- **Strong positive correlation** between Business Model and Vision/Moat (0.89)
- **Moderate correlation** between Team and Traction (0.53)
- **Negative correlation** between Problem and Market (-0.18), suggesting focus trade-offs.

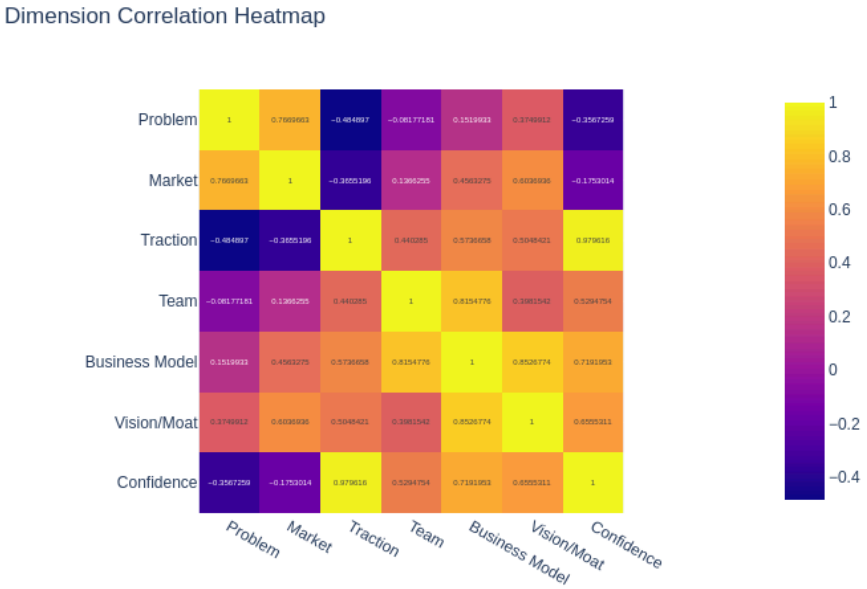


Fig 7.2 Correlation heatmap of dimensions

7.2.3 Word Cloud (Industry Distribution)

- **Social Media**: 2 companies (40%)
- **Ride-Sharing, B2C, Food Delivery**: 1 company each (20% each)

Fig 7.3 Word cloud from industry labels

7.2.4 Bar Chart Analysis

The bar chart visualization provides critical insights into overall pitch deck performance:

Performance Tiers Identified:

1. **Tier 1 - Market Leaders (70+ Score)**:
    - **Uber (77.0)**: Represents the gold standard with the highest final score
    - Demonstrates comprehensive pitch structure with minimal weaknesses
    - Strong across all dimensions, particularly team presentation and confidence
2. **Tier 2 - Viable Pitches (55-70 Score)**:
    - **Airbnb (64.1)**: Strong market opportunity focus but incomplete team section
    - **Facebook (60.3)**: Early-stage presentation with solid traction metrics
    - These pitches show potential but require targeted improvements
3. **Tier 3 - Needs Significant Work (<40 Score)**:
    - **DoorDash (24.4)** and **Social App (22.6)**: Both require major revisions
    - Multiple dimensional weaknesses across problem definition, market analysis, and team presentation
    - Represent common pitching mistakes that entrepreneurs should avoid

Score Distribution Insights:

- **54.4-point range** (22.6 to 77.0) indicates significant variability in pitch quality
- **Mean score of 49.7** suggests average pitch performance is below investor expectations
- **Large standard deviation (24.8)** reveals inconsistent pitch preparation across startups

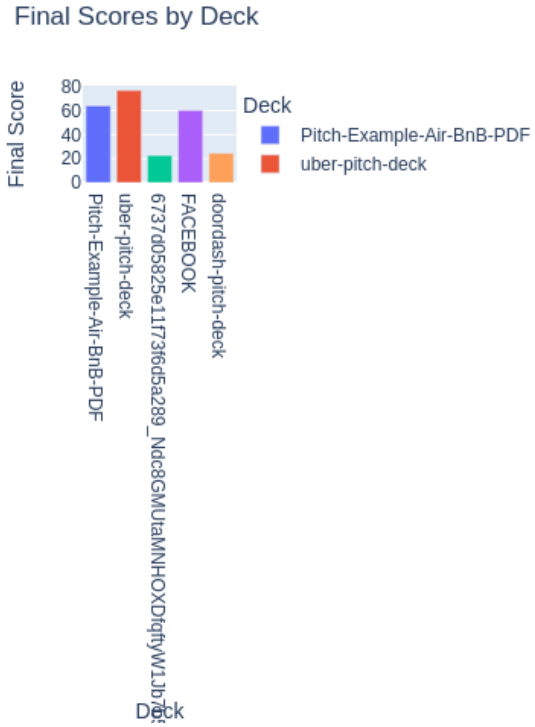


Fig 7.4 Bar chart of final scores

8. Inference

Analysis of 5 pitch decks reveals significant variations in scoring across different evaluation criteria, with final scores ranging from 23.7 to 74.0. The data suggests that certain industries and pitch elements are more critical for investor confidence than others.

8.1 Performance Analysis

8.1.1 Top Performers

1. **Uber (74.0/100)** - Ride-Sharing
    - Strongest areas: Team (9.4), Confidence (10.0), Traction (8.2)
    - Key success factor: Strong execution team with proven traction
2. **Airbnb (64.1/100)** - B2C
    - Strongest areas: Market (9.1), Confidence (9.4)
    - Key success factor: Large addressable market with clear value proposition
3. **Facebook (61.6/100)** - Social Media
    - Strongest areas: Traction (9.0), Confidence (10.0)
    - Key success factor: Early user adoption and network effects

8.1.2 Bottom Performers

- **Unnamed Social Media Deck (23.7/100)**: Scored poorly across all metrics except minimal confidence (2.0)
- **DoorDash (24.7/100)**: Despite strong traction (8.8), failed on problem definition and business model clarity

8.2 Critical Success Factors

8.2.1. Team Quality

- **High Impact**: Uber's team score of 9.4 contributed significantly to their top ranking
- **Major Gap**: Three decks scored 0.0 on team, indicating incomplete founder/team information
- **Inference**: Investors heavily weight team credibility and experience

8.2.2. Market Opportunity

- **Range**: 0.0 to 9.1, showing dramatic variation in market assessment
- **Top Scorer**: Airbnb (9.1) - large, underserved market
- **Inference**: Clear market size and opportunity articulation is fundamental

8.2.3. Problem Definition

- **Critical Weakness**: Two decks (Facebook and DoorDash) scored 0.0 on problem definition
- **Inference**: Many founders assume problem clarity without explicitly defining it

8.2.4. Traction vs. Other Metrics

- **Surprising Finding**: DoorDash had strong traction (8.8) but poor overall score (24.7)
- **Inference**: Traction alone cannot compensate for fundamental pitch weaknesses

8.3 Industry-Specific Patterns

8.3.1 Social Media (2 decks)

- **Performance**: Mixed (61.6 and 23.7)
- **Challenge**: Differentiation and network effect explanation
- **Pattern**: High confidence but inconsistent execution on fundamentals

8.3.2 B2C/Marketplace Models

- **Performance**: Strong (Airbnb 64.1, Uber 74.0)
- **Strength**: Clear value proposition for both sides of marketplace
- **Pattern**: Better at articulating market opportunity

8.3.3 Food Delivery

- **Performance**: Poor (DoorDash 24.7)
- **Issue**: Despite proven traction, failed to articulate problem and business model
- **Pattern**: Operations-heavy businesses struggle with pitch clarity

8.4 Clustering Analysis

8.4.1 Cluster 0 (High Performers)

- Uber, Facebook
- **Characteristics**: Strong fundamentals with good execution

8.4.2 Cluster 1 (Single Weakness)

- DoorDash
- **Characteristics**: Strong in some areas but critical gaps

8.4.3 Cluster 2 (Multiple Improvements Needed)

- Airbnb, Unnamed Social Media
- **Characteristics**: Require broad-based improvements

9. Future Work:

1. **Dynamic processing** would transform this from a proof-of-concept to a production-ready tool.
2. **Domain-specific fine-tuning** could dramatically improve scoring accuracy for pitch-specific language patterns.
3. **Interactive dashboard enhancements** would create genuine utility for investment professionals.